



# Singularity Overview

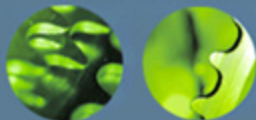
**James Larus**

Microsoft Research

(Galen Hunt & David Tarditi)

March 1, 2006

MSR TechFest

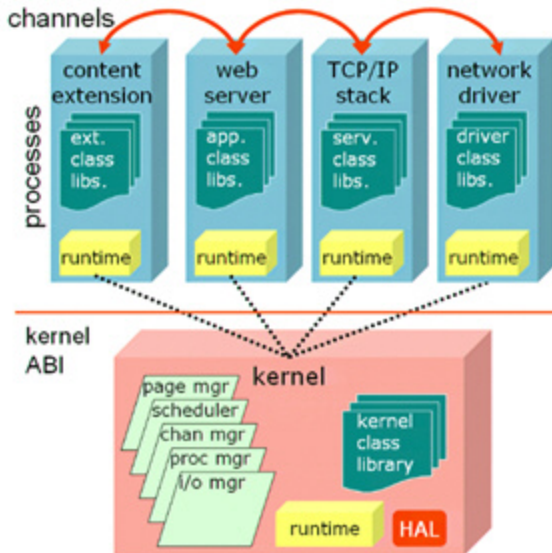


## Key Approaches

1. Pervasive use of safe (& analyzable) programming languages
  - type safety and memory safety
  - including device drivers, OS components, applications
2. Improve system resilience despite software errors
  - failure boundaries between components
  - improve extension model
  - explicit error notification
3. Increased verification
  - specification at multiple levels of abstraction
  - closed environments with explicit cross-domain interfaces
  - design for verifiability



# Singularity OS



- **Closed Kernel**

- 95% written in C#
  - 17% of files contain unsafe C#
  - 5% of files contain x86 or C++
- OS services & drivers in processes
- kernel closed at boot time

- **Software isolated processes (SIPs)**

- all user code is verified safe
- some unsafe code in trusted runtime
- processes closed at start time

- **Safe and efficient communication via strong interfaces**

- channels between processes
- channel behavior is specified & checked
- checked behavior enables efficient communication

- **Type safety is crux of verification & protection**



## Challenge 1: Pervasive Safe Languages

- Started with C#
  - actually Spec# (= C# + pre/post-conditions and invariants)
  - processes can be written in any safe language (with MSIL compiler)
- Added features to C# for systems programming
  - increase programmer control over allocation, initialization, and memory layout
- Explore language features to support programming and verification
  - message passing communication
  - factoring libraries
  - compile-time reflection





# Performance of Safe Languages

- Diverse requirements (not just speed)
  - execution time, memory footprint, pause times, expressibility
    - device drivers should have <100 KB dynamic footprint
    - kernel shouldn't fail due to transient memory exhaustion
    - able to write wide range of applications and devices
- JVM & CLR's design points not always appropriate
  - monolithic runtimes ("one size fits all")
    - powerful, general-purpose environments
    - large memory footprints (~4 MB per process for CLR)
    - long list of dependencies (CLR PAL requires >300 Win32 APIs)
  - JIT compilation
    - introduces complexity, performance overheads
  - runtimes replicate OS features
    - security, threading, configuration, etc.





## Small, Customizable, Safe Environment

- Singularity modular execution environment
  - runtime, garbage collector, and library selectable on per-process basis
    - reduce runtime overhead
    - enforce design discipline and system policies
  - lightweight, subsetable runtime
  - language support for factoring runtime and libraries
  - compiler support for specializing runtime and libraries
    - remove code for unused/disabled language features
- Ahead-of-time optimizing compilation (MSR Bartok)
- Merge OS & language run time
  - integrate partially redundant features into OS
  - no separate JVM or CLR





## Runtime Overhead

	Memory footprint "Hello World" process			
	Singularity	FreeBSD 5.3	Linux 2.6.11 (Red Hat FC4)	Windows XP (SP2)
C - static lib		232K	664K	544K
C++ - static lib		704K	1,216K	572K
C# - w/ GC	408K*			3,750K

- C# process w/ GC has similar memory footprint to C++
  - minimal process (no GC or exceptions) is ~16K





## Challenge 2:

# Run-Time Resilience

- Software errors should not cause system failure
- Resilient system architecture
  - isolate system components to prevent data corruption
  - provide clear failure notification
  - implement policy for restarting failed component

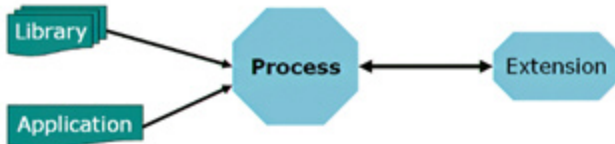






# Extensibility Through Dynamic Code Loading

- Code loading is common extensibility mechanism
  - DLLs, Java class loading, browser plug-ins, Eclipse, ...
- Shared state reduces dependability
  - 85% of Windows crashes are device drivers
    - problem at all levels, not just the OS
  - no isolation boundary between code and extension
- Singularity processes are fundamental unit of failure isolation
  - no dynamic code loading or run-time code generation
    - all code present when process starts execution
  - any extension executes in separate process
    - two closed environments with well-defined interface
  - no shared memory





# Making Processes Lightweight

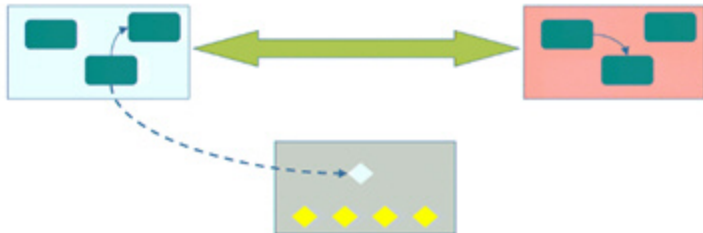
- Hardware context switch mechanisms can be expensive
  - use of kernel calls, IPC limited by their cost
- Software Isolated processes (SIPs)
  - processes execute only safe or trusted code
  - process is exclusively owner of a set of pages
  - protection and isolation enforced by language safety and kernel ABI design (not hardware)
  - global invariants:
    - no process contains a pointer to another object space
    - no pointers from exchange heap into process
- Closed object space (not address space)





# Fast Interprocess Communication

- Channels
  - bi-directional, asynchronous IPC mechanism between two processes
  - contract specifies messages and communication protocol
- "Send & forget" message semantics
  - each message and endpoint owned by at most one process
  - ownership transferred at send
- Correct usage statically enforced
- Efficient implementation





## Failure Isolation

- SIPs are failure containers
  - no shared implementation or state across SIPs
  - process runtimes are distinct
- On SIP failure:
  - clean failure notification on peer channel endpoints
  - resources reclaimed by OS
- Recovery feasible, not automatic or transparent
  - peers can recover and continue



## Micro Benchmarks

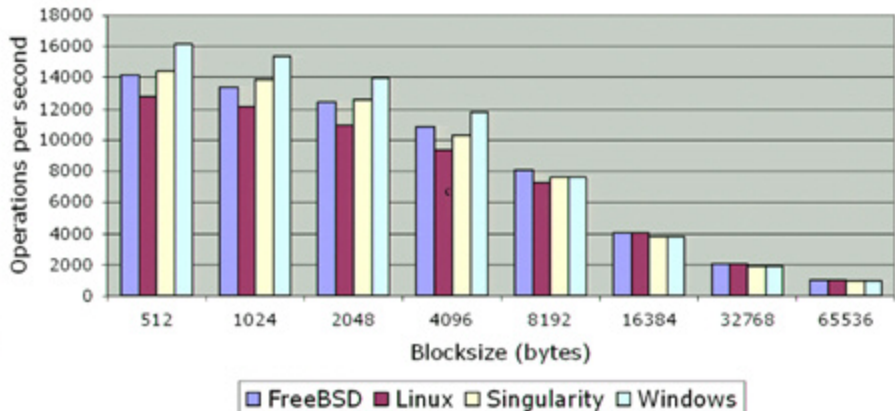
Athlon64 3000+ (1.8GHz) nForce4 SLI	Cost (CPU Cycles)			
	Singularity	FreeBSD 5.3	Linux 2.6.11 (Red Hat FC4)	Windows XP (SP2)
Minimum kernel API call	87	878	437	627
Message request/reply	1,450	13,300	5,800	(LPC) 4,650 (NP) 6,340
Process create & start	300,000	1,030,000	719,000	5,380,000

- Why?
  - all SIPs run in ring 0
  - static verification replaces hardware protection
  - good optimizing compiler (not JIT)



# I/O Throughput

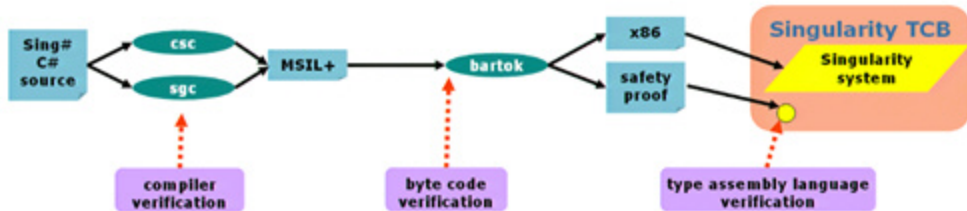
**Sequential Read Performance**





### Challenge 3: More Verification

- Integrate specifications throughout system
  - language
  - interprocess communication
  - application & device driver descriptions
- Detect errors early, verify code late
  - language safety essential to system integrity





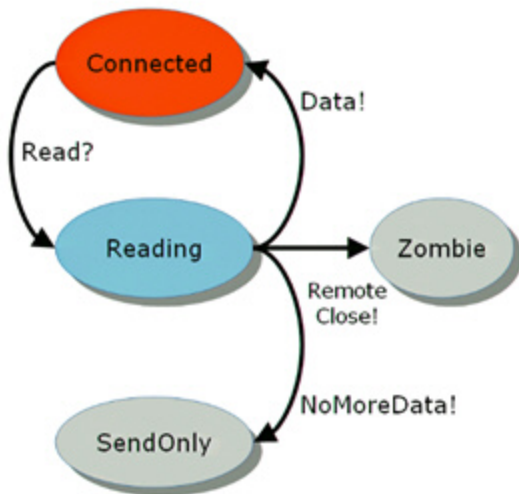


Example:

# Channel Contracts

```
public contract TcpSocketContract {  
    ...  
    state connected : one {  
        Read? -> ReadResult;  
        Write? -> WriteResult;  
  
        GetLocalAddress? -> IPAddress! ->  
        Connected;  
        GetLocalPort? -> Port! -> Connected;  
  
        DoneSending? -> ReceiveOnly;  
        DoneReceiving? -> SendOnly;  
        Close? -> closed;  
        Abort? -> closed;  
    }  
  
    state Reading : one {  
        Data! -> Connected;  
        NoMoreData! -> SendOnly;  
        RemoteClose! -> Zombie;  
    }  
    ...  
}
```

? = receive  
! = send





Example:

# Channel Contracts

Contract

Client

```
public contract TcpConnectionContract {  
    ...  
    state Connected : one {  
        Read? -> ReadResult;  
        Write? -> WriteResult;  
  
        GetLocalAddress? -> IPAddress! ->  
        Connected;  
        GetLocalPort? -> Port! -> Connected;  
  
        DoneSending? -> ReceiveOnly;  
        DoneReceiving? -> SendOnly;  
        Close? -> Closed;  
        Abort? -> Closed;  
    }  
  
    state Reading : one {  
        Data! -> Connected;  
        NoMoreData! -> SendOnly;  
        RemoveClose! -> Zombie;  
    }  
    ...  
}
```

```
...  
conn.SendRead();  
switch receive {  
    case conn.Data(readData) :  
        dataBuffer.AddToTail(readData);  
        return true;  
  
    case conn.RemoteClose() :  
        return false;  
}  
...
```



Example:

## Applications Specifications

- Application is first-class abstraction with identity
  - code + resources + manifest
- Manifest specifies
  - software components
  - dependencies
  - exported channels
  - hardware or software resource requirements



# Device Driver Specification

requires PCI Device

```
[DriverCategory]
[Signature("/pci/03/00/5333/8811")]
class S3Trio64Config : DriverCategoryDeclar
```

requires 4MB frame buffer  
(declared in PCI config)

```
{
  [IoMemoryRange(0, Length = 0x400000)]
  IoMemoryRange framebuffer;
```

requires system  
console buffer

```
  [IoFixedMemoryRange(Base = 0xb8000, Length = 0x8000)]
  IoMemoryRange textBuffer;
```

requires VGA I/O  
ports

```
  ...
  [IoFixedPortRange(Base = 0x3c0, Length = 0x20)]
  IoPortRange control;
```

requires channel to  
parent process for  
control

```
  [ExtensionEndpoint(typeof(ExtensionContract.Exp))]
  TRef<ExtensionContract.Exp:Start> pnp;
```

```
  [ServiceEndpoint(typeof(VideoDeviceContract.Exp))]
  TRef<ServiceProviderContract.Exp:Start> video;
```

exports channel for  
clients to access  
video device

```
  ...
}
```



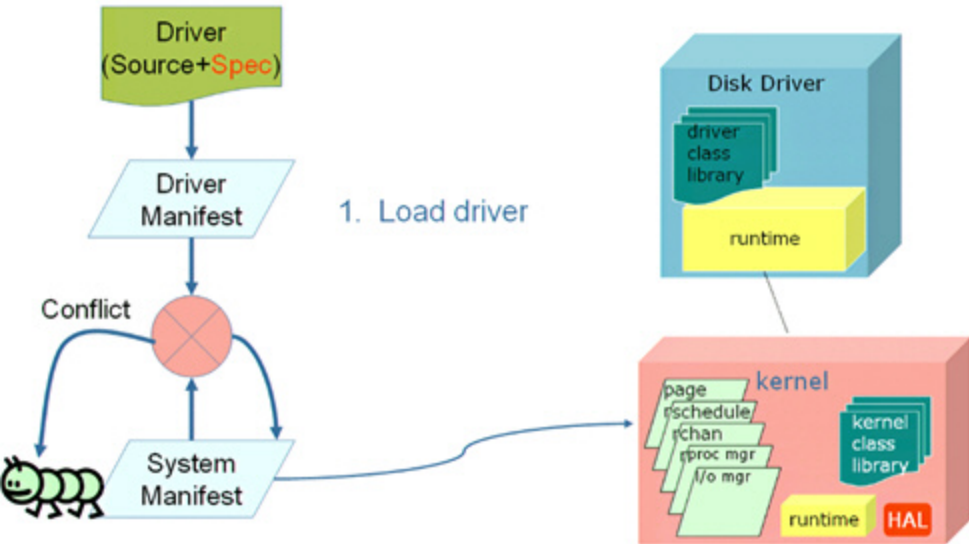
# Specification Used Many Ways

Driver  
(Source+Spec)



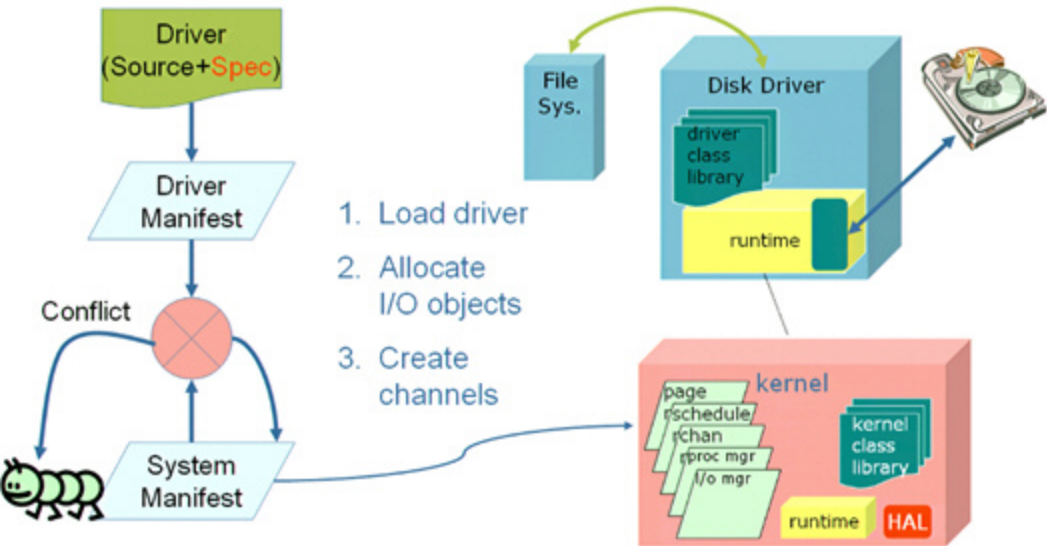


# Specification Used Many Ways





# Specification Used Many Ways







## Driver Verification

- Verification ensures
  - never install an program that will break another program
  - never start a program without appropriate resources
  - never grant a program access to undeclared resources
- All of these checks performed statically



## Summary

- Singularity is basis for more dependable systems
  - pervasive use of safe programming languages
  - lightweight, closed, customizable run-time environment
  - verifiable specification of system behavior
- Working research prototype
  - driving research in large number of areas
- More information:
  - <http://research.microsoft.com/os/singularity>  
MSR Tech Report MSR-TR-2005-135